

Computer Science Department

TECHNICAL REPORT

Deterministic Coin Tossing With Applications to
Optimal Parallel List Ranking

by

Uzi Vishkin†
(Tel Aviv University)
Richard Cole‡
(New York University)

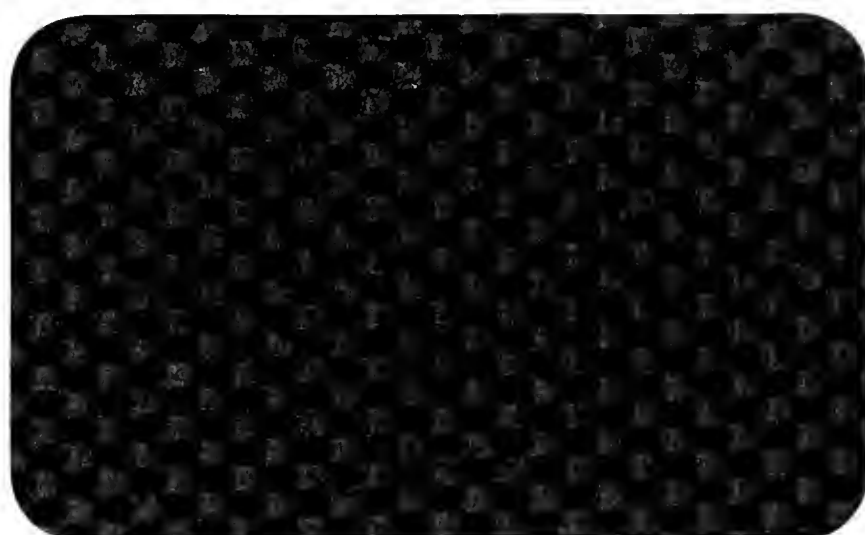
Technical Report #175
Ultracomputer Note #89
September, 1985

NEW YORK UNIVERSITY



Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET NEW YORK, N.Y. 10011

NYU COMPSCI TR-175
Vishkin, Uzi
Deterministic coin tossing
with applications to
optimal parallel list



Deterministic Coin Tossing With Applications to
Optimal Parallel List Ranking

by

Uzi Vishkin†
(Tel Aviv University)
Richard Cole‡
(New York University)

Technical Report #175
Ultracomputer Note #89
September, 1985

New York University
Dept. of Computer Science
Courant Institute of Mathematical Sciences
251 Mercer Street
New York, New York 10012

†This research was supported by NSF grants NSF-DCR-8318874, NSF-DCR-8413359, NSF-DCR-84-01633, ONR grant N0014-85-K-0046 and by the Applied Mathematical Sciences subprogram of the office of Energy Research, U.S. Department of Energy under contract number DE-AC02-76ER03077. Present address for Uzi Vishkin: Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, ISRAEL.

‡This research was partially supported by NSF Grant DCR-84-01633 and by an IBM faculty development award.

ABSTRACT

The following problem is considered: given a linked list of length n , compute the distance from each element of the linked list to the end of the list. The problem has two standard deterministic algorithms: a linear time serial algorithm, and an $O(\log n)$ time parallel algorithm using n processors. We present new deterministic parallel algorithms for the problem. Our strongest results are:

1. $O(\log n \log^{(k)} n)$ time using $n/(\log n \log^{(k)} n)$ processors, for any fixed positive integer k , where $\log^{(k)}$ is the k -th iterate of the log function. This algorithm achieves optimal speed-up.
2. $O(\log n \log^* n)$ time using $n/\log n$ processors. Since $\log^* n$ grows extremely slowly as a function of n this algorithm achieves optimal speed-up for all practical purposes.
3. $O(\log n)$ time using $n \log^{(k)} n / \log n$ processors, for any fixed positive integer k .

The algorithms apply a novel "random-like" deterministic technique. This technique provides for a fast and efficient breaking of a symmetric situation in parallel.

1. Introduction

The model of parallel computation used in this paper is the exclusive-read exclusive-write (EREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes. See [Vi-83a] for a survey of results concerning PRAMs.

Let $Seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem being considered. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal*.

We present a new deterministic *coin tossing* technique for devising parallel algorithms. The technique uses the binary representation of names (numbers) for breaking a symmetric situation in a "random-like" fashion.

Let m be the size of the memory of our computer. Our technique performs well when each variable in the underlying model of computation is represented by a few bits (say $O(\log m)$ bits). Interestingly, the technique performs badly when each variable is represented by many bits (say $f(m)$ bits, where f is the inverse of \log^*). Representing each variable by $O(\log m)$ bits is in line with typical definitions of RAMs (see [AHU-74]). The role of PRAMs is to extend the RAM model to express parallelism. This extension

should have no effect on the number of values that each variable may assume. A variant of PRAMs (called PRAM-INFINITY) that allows each variable to assume infinitely many values has been proposed recently. The PRAM-INFINITY also allows infinitely large shared memory. This variant (or closely related ones) was used to prove lower bounds for various interesting problems; the proofs apply mathematically appealing "Ramsey-like" theorems (see [FMRW-85], [IM-85], [MW-85]).

It appears that in the transition from PRAM to PRAM-INFINITY we lose the coin tossing technique. For the technique depends crucially on the fact that each variable is represented by few bits (say $O(\log m)$ bits), while in the PRAM-INFINITY model this constraint does not exist; in fact, there is no restriction on the number of bits representing a variable. This is analogous to the loss of bucket sort when we adopt the decision tree model. (See [AHU-74] for both an $\Omega(n \log n)$ time lower bound for sorting n elements in a decision tree model and an $O(n)$ time bucket sort algorithm).

We show how to apply our coin tossing technique to the *list-ranking problem* defined below.

Input: A linked list of length n . It is given in an array of length n , not necessarily in the order of the linked list. Each of the n elements (except the last element in the linked list) has the array index of its successor in the linked list.

The problem: For each element, compute the number of elements following it in the linked list.

The list ranking problem is encountered often in the design of parallel algorithms. For instance, the fundamental "Euler tour technique" for computing various tree functions (see [TV-83] and [Vi-85]) has the same efficiency as the new algorithm presented here.

The problem has a trivial linear time serial algorithm and a simple deterministic parallel algorithm. This parallel algorithm runs in time $O(\log n)$ using n processors. However, Wyllie [W-79] conjectured that $\Omega(n)$ processors are required in order to get $O(\log n)$ time. If true, this would imply, in particular, that there is no optimal speed-up parallel algorithm for $n/\log n$ processors. [KRS-85] recently presented an optimal speed-up algorithm for this problem that runs in $O(n^\epsilon)$ time using $n^{1-\epsilon}$ processors, for fixed ϵ , $1 \geq \epsilon > 0$. [Vi-84b] proposed to use randomized parallel algorithms for this problem. A randomized parallel algorithm which runs in $O(n/p)$ time using $p \leq n/(\log n \log^* n)$ processors on an EREW PRAM was given. The probability that this will indeed be the running time converges rapidly to one as n grows. In particular, this optimal speed-up algorithm runs in "about" $O(\log n)$ time using "about" $n/\log n$ processors.

In this paper we present new deterministic parallel algorithms. Our strongest results are:

1. $O(\log n \log^{(k)} n)$ time using $n/(\log n \log^{(k)} n)$ processors, for any fixed positive integer k , where $\log^{(k)}$ is the k -th iterate of the log function. This algorithm achieves optimal speed-up.
2. $O(\log n \log^* n)$ time using $n/\log n$ processors. Recall that $\log^* n$ grows extremely slowly and can be viewed as a constant for all practical purposes. (For instance, $\log^* 2^{65536} = 5$. See the function G in [AHU-74], p. 133). Therefore, we can justifiably say that our algorithm achieves optimal speed-up for all practical purposes.
3. $O(\log n)$ time using $n \log^{(k)} n / \log n$ processors, for any fixed positive integer k , thereby showing that Wyllie's conjecture is incorrect.

The next section presents the new deterministic coin tossing technique for breaking a symmetric situation. Among other things, Section 3 reviews an optimal speed-up deterministic parallel algorithm that uses balanced trees. The algorithm is used later for two purposes: (1) as a subroutine, and (2) to explain the new list ranking algorithm. The new algorithm essentially grafts the new technique onto the framework of the balanced tree algorithm. In section 4 we describe the basic version of our algorithm that runs in time $O(\log n \log \log n)$ using $n \log^* n / (\log n \log \log n)$ processors. This algorithm achieves close to optimal speed-up; it will be quite adequate for all practical purposes. In section 5 we describe the algorithms that achieve optimal speed-up and our other results.

2. The deterministic coin tossing technique

We illustrate the deterministic coin tossing technique by using it to break the symmetric situation that arises in the following problem.

Input: A connected directed graph $G(V, E)$. The in-degree of each vertex is exactly one. The out-degree of each vertex is exactly one. Such a graph is called a *ring* since it forms a directed circuit. Let $n = |V|$.

We define a subset U of V to be an r -ruling set of G if:

- (1) No two vertices of U are adjacent.
- (2) For each vertex v in V there is a directed path from v to some vertex in U whose edge length is at most r .

The r -ruling set problem: Find an r -ruling set of V .

In order to demonstrate our basic technique we give an $O(1)$ time algorithm using n processors for the $\lceil \log n \rceil$ -ruling set problem. The algorithm is given for the EREW PRAM.

Later, we present a recursive application of the technique. It leads to an $O(k)$ time algorithm using n processors for the $\lceil \log^{(k)} n \rceil$ -ruling set problem. In particular, it provides an $O(\log^* n)$ time algorithm using n processors for the 2-ruling set problem.

Assumptions about the input representation: The vertices are given in an array of length n . The entries of the array are numbered from 0 to $n-1$. The numbers are represented as binary strings of length $\lceil \log n \rceil$. We refer to each binary symbol (bit) of this representation by a number between 0 and $\lceil \log n \rceil - 1$. The rightmost (least significant) bit is called bit number 0 and the leftmost bit is called bit number $\lceil \log n \rceil - 1$. Each vertex has a pointer to the next vertex in the ring (representing its outgoing edge). For simplicity we assume that $\log n$ is an integer¹.

Here is a verbal description of an algorithm for the $\log n$ -ruling set problem. The algorithm is given later. Processor i , $0 \leq i \leq n-1$, is assigned to entry i of the input array (for simplicity, entry i is called vertex i). It will attach the number i to vertex i . So, the *present* "serial" number of vertex i , denoted $SERIAL_0(i)$, is i . Next, we attach to vertex i a new serial number, denoted $SERIAL_1(i)$, as follows. Let i_1 be the vertex preceding i and let i_2 be the vertex following i . (That is (i_1, i) and (i, i_2) are in E). Let j be "the number of the rightmost bit in which i and i_2 differ". Processor i assigns j to $SERIAL_1(i)$.

Example. Let i be ...010101 and i_2 be ...111101. The number of the rightmost bit in which i and i_2 differ is 3 (recall the rightmost bit has number 0). Therefore, $SERIAL_1(i)$ is 3.

Remark (Due to B. Schieber). j can be computed by a constant number of standard operations, as follows. Without loss of generality suppose $i \geq i_2$ (otherwise interchange the two numbers). Set $h = i - i_2$, and $k = h - 1$. (So h has a 1 for bit number j , and a 0 for bits of lesser significance, while k has a 0 for bit number j , and a 1 for bits of lesser significance; also, h and k agree on the bits of higher significance.) Compute $l = h \odot k$, where \odot is the exclusive-or operation. We observe l is the unary representation of $j+1$. So it just remains to convert this value from unary to binary, and then to subtract one.

Next, we show how to use the information in vector $SERIAL_1$ in order to find a $\log n$ -ruling set.

Fact 1: For all i , $SERIAL_1(i)$ is a number between 0 and $\log n - 1$ and needs only $\lceil \log \log n \rceil$ bits for its representation. For simplicity we will assume that $\log \log n$ is an integer.

Fact 2: Suppose $SERIAL_1(i)$ is a local minimum. (That is, $SERIAL_1(i) \leq SERIAL_1(i_1)$ and $SERIAL_1(i) \leq SERIAL_1(i_2)$, where i_1 is the vertex preceding i , and i_2 is the vertex following i .) Then:

¹The base of all logarithms in the paper is 2.

- (a) At least one of $SERIAL_1(i)$, $SERIAL_1(i_2)$, and $SERIAL_1$ of the $\log n - 2$ vertices immediately further beyond i_2 is a local maximum.
- (b) Let k be such a local maximum that is closest to i . Then, at least one of $SERIAL_1(k)$ and $SERIAL_1$ of the $\log n - 1$ vertices immediately further beyond k is a local minimum.
- (c) The length of the shortest path from any vertex in G to a (vertex which provides a) local extremum (minimum or maximum), with respect to $SERIAL_1$, is at most $\log n - 1$.

Observe that several local minima (or maxima) may form a "chain" of successive vertices in G . Requirement (1), in the definition of an r -ruling set, does not allow us to include all these local minima in the set of selected vertices. Our algorithm exploits the alternation property (defined below) of vector $SERIAL_1$ to overcome this problem.

The alternation property: Let i be a vertex and j be its successor. If bit number $SERIAL_1(i)$ of $SERIAL_0(i)$ is 0 (resp. 1), then this bit is 1 (resp. 0) in $SERIAL_0(j)$.

Suppose that $i_1, i_2 \dots$ is a chain in G such that $SERIAL_1(i)$ is a local minimum (resp. maximum) for every i in the chain. Then:

Fact 3: For all vertices in the chain $SERIAL_1$ is the same (i.e., $SERIAL_1(i_1) = SERIAL_1(i_2) = \dots$). (By definition of local minimum).

Below, we consider bit number $SERIAL_1(i_1)$ of $SERIAL_0$ for all vertices in the chain. Let i_l, i_{l+1} be two adjacent vertices in the chain.

Fact 4: Bit number $SERIAL_1(i_1)$ of $SERIAL_0(i_l)$ is not equal to bit number $SERIAL_1(i_1)$ of $SERIAL_0(i_{l+1})$. (This is readily implied by the alternation property).

We select all vertices i that are local minima and satisfy one of the following two conditions:

- (1) Neither of i 's neighbors (the vertices adjacent to i) is a local minimum.
- (2) Bit number $SERIAL_1(i)$ is 1.

We say an unselected vertex is *available* if neither of its neighbors was selected and it is a local maximum. We select all available vertices i that satisfy one of the following two properties.

- (1) Neither of i 's neighbors is available.
- (2) Bit number $SERIAL_1(i)$ is 1.

The selected vertices form a $\log n$ -ruling set. Requirement (1) is satisfied since we never select two adjacent vertices. Requirement (2) is satisfied by Fact 2(c) and since every local extremum is either selected or is a neighbor of a vertex that was selected.

Less informally we write the algorithm as follows. (Later, we will refer to this as the basic step.)

```

for Processor  $i$ ,  $0 \leq i \leq n-1$ , pardo
   $SERIAL_0(i) := i$ 
   $SERIAL_1(i) :=$  "the minimal bit in which  $SERIAL_0(i)$  differs from
     $SERIAL_0$  of the following vertex"
  if  $SERIAL_1(i)$  is a local minimum with respect to the two neighbors of  $i$ 
  then if either of the following is satisfied:
    (1) neither of the vertices adjacent to  $i$  is a local minimum
    (2) bit number  $SERIAL_1(i)$  of  $SERIAL_0(i)$  is 1
    then select  $i$ 
  if neither  $i$  nor any of its neighbors were selected and if  $SERIAL_1(i)$  is
    a local maximum with respect to the two neighbors of  $i$ 
  then (**  $i$  is available, and **) if either of the following is satisfied:
    (1) neither of the vertices adjacent to  $i$  is available
    (2) bit number  $SERIAL_1(i)$  of  $SERIAL_0(i)$  is 1
    then select  $i$ 

```

Below, we show how to apply the basic step repeatedly in order to find a 2-ruling set.

The k -th application of the basic step.

In order to prepare the input for the k -th application of the basic step, we "delete" from G the vertices that were selected in the previous $k-1$ applications, their neighbors, and the edges incident to any vertex being deleted.

The input for the k -th application of the basic step is the remaining graph and vector $SERIAL_{k-1}$. $SERIAL_{k-1}$ will play the role played above by $SERIAL_0$ and a new vector $SERIAL_k$ will play the role of $SERIAL_1$. The degree of each vertex in the input graph is at most 2 (if the directions of the edges are ignored). It will be very simple to extend the basic step to handle vertices whose degree is ≤ 1 . Vertices whose degree is 2 are treated as in the basic step (unless they have a neighbor whose degree is 1). The k -th application of the basic step will be as follows. (For an explanation see Fact 5 below.)

```

for processor  $i$ ,  $0 \leq i \leq n-1$ , pardo
  if vertex  $i$  or one of its neighbors have been selected
    in a previous application of the basic step
  then "delete" vertex  $i$  and the edges incident to it
for processor  $i$ ,  $0 \leq i \leq n-1$ , such that  $i$  is in the remaining graph pardo
  case 1  $deg(i) = 2$ 
    then compute  $SERIAL_k(i)$ 
    if the degree of each of  $i$ 's two neighbors is 2
    then apply the basic step to  $i$ 
  case 2  $deg(i) = 0$ 

```

```

    then select  $i$ 
case 3  $\deg(i) = 1$ 
    then if either of the following is satisfied
        (1) the degree of  $i$ 's neighbor is 2
        (2)  $i$ 's neighbor is its successor
    then select  $i$ 

```

The following fact helps to clarify the operation of the k -th application of the basic step.

Fact 5: Let i, j be adjacent in the input graph for the k -th application. Then:

$SERIAL_{k-1}(i) \neq SERIAL_{k-1}(j)$. (For $k=1$ this inequality clearly holds. We show that it also holds if $k > 1$. If they were equal each of them had to be a local maximum or local minimum at the $(k-1)$ -st application. The selection of the ruling set implies that each local maximum or local minimum is either selected or has a neighbor being selected. Therefore, it must have been deleted and cannot be included in this input graph).

Fact 6: It is easy to deduce that the output graph consists of simple paths each of length at most $\log \log \cdots \log n - 1$ (counting edges) where the sequence includes k "log"s. (Again, we assume for simplicity that each application of a sequence of logs to n produces only integers).

We finish this description with three obvious conclusions.

- (1) After a total of $\log^* n$ applications we delete all vertices in the graph.
- (2) The vertices that were selected form a 2-ruling set.
- (3) The cardinality of a 2-ruling set (in a ring) is at least $n/3$.

If our original input is a directed path of n vertices, rather than a ring, we obtain a 2-ruling set by applying the basic step $\log^* n$ times, as above. To obtain a $\log^{(k)} n$ -ruling set we apply the basic step k times.

General remarks.

1. Readers familiar with randomized algorithms may be tempted to solve these problems using randomization. We already mentioned that [Vi-84b] did so for (the related) list ranking problem. Our deterministic technique was inspired by such a randomized approach.
2. The $\lceil \log n \rceil$ -ruling set algorithm is valid even for models of distributed computation that allow only local communication and do not have a shared memory like a PRAM. We do not elaborate on this.

3. Balanced tree algorithms

3.1. Preliminaries

Theorem (Brent). Any synchronous parallel algorithm taking time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lceil x/p \rceil + t$.

Proof of Brent's theorem. Let x_i denote the number of operations performed by the algorithm in time i ($\sum_1^t x_i = x$). We use the p processors to "simulate" the algorithm. Since all the operations at time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_1^t \lceil x_i / p \rceil \leq \sum_1^t (x_i / p + 1) \leq \lceil x/p \rceil + t . \square$$

Remark. The proof of Brent's theorem poses two implementation problems. The first is to evaluate x_i at the beginning of time i in the algorithm. The second is to assign the processors to their jobs.

Recall the following standard *deterministic parallel algorithm* for the list-ranking problem (defined in the Introduction). Say that we have n processors. Assign a processor to each of the n elements. Denote the pointer of element i of the input array by $D(i)$ and initialize $R(i) := 1$, $1 \leq i \leq n$. We set $D(t) :=$ "end of list" (where t is the last element in the linked list), $D(\text{"end of list"}) :=$ "end of list" and $R(\text{"end of list"}) := 0$.

Iterate $\lceil \log n \rceil$ times:

for processor i , $1 \leq i \leq n$, pardo (perform in parallel)

$R(i) := R(i) + R(D(i)); D(i) := D(D(i))$ (To be called the short-cut operation, performed by i at $D(i)$). (See Fig. 2.)

Note that $\Omega(n \log n)$ short-cuts are made by this algorithm. It runs in time $O((n \log n)/p + \log n)$ using p processors on an EREW PRAM and solves the list ranking problem, by placing the results in the vector R .

Implementation Remark 1. In order to derive this running time from Brent's theorem n has to be broadcast to all p processors. This takes an additional $O(\log p)$ time.

Implementation Remark 2. As presented the algorithm is not EREW since there are concurrent reads at "end of list". This can be avoided by instructing every processor i to quit when $D(i) =$ "end of list".

3.2. Balanced binary tree parallel algorithms.

One simple pattern of optimal speed-up deterministic parallel algorithms uses the balanced binary tree. This pattern was used, among many others, by [W-79], [CLC-81] and [Vi-84a]. Let us first demonstrate this pattern on the

problems of computing sums and prefix sums.

Input: An array of n numbers $A(1), A(2), \dots, A(n)$. Assume, without loss of generality, that $\log_2 n$ is an integer.

Problem: Compute their sum.

Algorithm: "Plant" a balanced binary tree with n leaves on the array. The nodes of the tree at level h are denoted $[h, j]$, $1 \leq j \leq 2^{\log n - h}$. See Fig. 3. Leaf $[0, j]$ corresponds to $A(j)$. Associate a number $B[h, j]$ with node $[h, j]$ of the tree.

Initialization: for all $1 \leq j \leq n$ pardo $B[0, j] := A(j)$.

for $h := 1$ to $\log n$ do

 for all $1 \leq j \leq 2^{\log n - h}$ pardo $B[h, j] := B[h-1, 2j-1] + B[h-1, 2j]$.

$B[\log n, 1]$ holds the desired sum.

Think first about an n processor implementation of this summation algorithm. It runs in $O(\log n)$ time. Then apply the proof of Brent's Theorem to get an alternate implementation that uses only $n/\log n$ processors and runs in $O(\log n)$ time. This summation algorithm can be extended to solve the following prefix sum problem.

Input: Same as for the summation problem.

Problem: Compute $\sum_{j=1}^i A(j)$ for all $1 \leq i \leq n$.

Algorithm: Perform the summation algorithm given above, thereby obtaining all the B values. An additional "down-sweep" of the tree (from the root to the leaves), which roughly amounts to reversing the operation of the summation algorithm, will complete the job.

Associate another number $C[h, j]$ with each node $[h, j]$.

Initialization: $C[\log n, 1] := 0$.

for $h := \log n - 1$ downto 0 do

 for all $1 \leq j \leq 2^{\log n - h}$ pardo

 if j is odd

 then $C[h, j] := C[h+1, (j+1)/2]$

 else $C[h, j] := C[h+1, j/2] + B[h, j-1]$.

for all $1 \leq j \leq n$ pardo $C[0, j] := C[0, j] + B[0, j]$.

$C[0, j]$, $1 \leq j \leq n$, hold the desired prefix sums. This algorithm can also be implemented to run in $O(n/p + \log n)$ time using p processors on an EREW PRAM. (Apply Brent's theorem and Implementation Remark 1.)

A wishful thought. We want to find an algorithm for the list ranking problem that performs a total of $O(n)$ short-cuts. If we could "plant" a balanced binary tree in our linked list (in the order of the linked list) it would solve

our problem: enter a one at each leaf and apply the prefix sum algorithm. A closer look at the summation part of such a prefix sum computation reveals the following:

The operation of the **for** statement (of the summation algorithm) for $h=1$ corresponds to short-cuts at every odd location in the linked list. This results in a new linked list that connects only the even locations of the original list, thereby halving its length. Then, the **for** statement for $h=2$ corresponds to short-cuts at odd locations of the new linked list, and so on. See Fig. 4. Observe that the **for** statement of the summation algorithm never performs a short-cut at two successive elements of the linked list at hand; and, therefore, the "input" to any operation of this **for** statement is a single linked list.

Remark: The problem, of course, is that we do not know how to plant a balanced binary tree with respect to the linked list without actually first solving the list ranking problem itself, since this "planting" needs the ranking mod 2, mod 4, mod 8,... as explained above.

Each operation of the **for** statement has the following two features.

- (1) The output is a single list whose length is half the length of the input.
- (2) It takes $O(1)$ parallel time to execute.

We will use an algorithm which approximates these two features. In our new algorithm we plant an "approximately balanced tree" (it will be a 2-3 tree). Each leaf of the tree corresponds to an element of the list, and each level of the tree corresponds to an iteration of the **for** statement. For a given level of the tree, the nodes at this level correspond to those elements of the list over which shortcuts have not yet been made (by iterations of the **for** statement corresponding to lower levels of the tree). For each level of the tree we divide the elements of the list (corresponding to nodes at this level) into two sets: those that are shortcut (by the corresponding iteration of the **for** statement), called *victims*, and those that are not shortcut (called *survivors*). In order to approximately achieve properties (1) and (2) above, we require these two sets to meet the following two constraints:

- (a) If an element is a survivor then its successor (if any) is a victim.
- (b) One, at least, of every three adjacent elements is a survivor.

By (a) at most one half of the elements are survivors. By (b) each survivor need perform at most two shortcut operations to remove all the victims from the list. Hence in $O(1)$ parallel time (using n processors) we obtain a single linked list containing at most half as many elements (assuming we can separate the elements into survivors and victims).

But a 2-ruling set provides an appropriate set of survivors!

4. The basic list ranking algorithm

Initialization: $m := n$. As in the standard deterministic algorithm, denote the pointer of element i by $D(i)$ and initialize $R(i) := 1$, $0 \leq i \leq n-1$.

The algorithm which is given later should be read together with the commentary below. The purpose of the **while** loop of the algorithm is to "thin out" the input linked list into a list of length $\leq n/\log n$. The input to each iteration of the **while** loop is a linked list of length m stored in an array of length m . Vector D contains, for each element, the next element in this linked list.

The purpose of Step 2 is to enter either the value 1 or the value 0 into $RULING(j)$, for each j , $0 \leq j \leq m-1$, so that those elements with $RULING(j) = 1$, $1 \leq j \leq m$, form a 2-ruling set of the directed graph. Step 2 uses essentially the algorithm of Section 2 for finding a 2-ruling set.

In Step 3 we shortcut, in parallel, over each j such that $RULING(j) = 0$. The resulting list will contain exactly those elements in the 2-ruling set; of which there are at most $m/2$. We make some further comments on the operation of this step.

(a) Each element j for which $RULING(j)=1$ (an element of the 2-ruling set) is followed by at least one and at most two elements for which $RULING$ is 0.

(b) Each element over which we perform a shortcut will remain with no incoming pointers. Such elements will be "deleted" in Step 4.

(c) The parameter t stands for the present time. The information in $OP(i,t)$ enables us, later on, to reconstruct the operation of processor i at time t . This is used in Step 6 to derive the final value of $R(D(j))$ by subtracting the present value of $R(j)$ from the final value of $R(j)$. For this reason we preferred here to name the processors performing the operations rather than to use the framework of Brent's theorem.

Step 4 contracts the input array for the present **while** loop iteration into a new array that contains exactly those elements in the new linked list.

When we arrive at Step 5, the length of the linked list at hand is $\leq n/\log n$. Step 5 applies the standard list ranking algorithm in order to find the ranking of each element in this linked list.

Step 6 extends the list rankings to all elements of the original linked list using the information in $OP(.,.)$.

while $m > n/\log n$ **do**

Step 1. (Initialization for the present **while** loop iteration).

for j , $0 \leq j \leq m-1$, **pardo**

$SERIAL_0(j) := j$

Step 2. Compute a 2-ruling set into vector $RULING$.

 From now on we specify for each instruction the processors that perform it. Suppose p processors are available. Processor i , $1 \leq i \leq p$, is assigned to segment $[(i-1)m/p, \dots, im/p - 1]$ of the array that forms the input to

this whileloop iteration. (For simplicity we assume that m/p is an integer. Otherwise, we could assign Processor i to the segment including all the integers in the half open interval $((i-1)m/p - 1; im/p - 1]$.)

Step 3.

```

for Processor  $i$ ,  $1 \leq i \leq p$ , pardo
  for  $j := (i-1)m/p$  to  $im/p - 1$  do
    if  $RULING(j) = 1$ 
      then  $OP(i,t) := (D(j),j,R(j));$ 
            $R(j) := R(j) + R(D(j))$  ;  $D(j) := D(D(j))$  (shortcut).
           if  $RULING(D(j)) = 0$ 
             then  $OP(i,t) := (D(j),j,R(j));$ 
                   $R(j) := R(j) + R(D(j))$  ;  $D(j) := D(D(j))$  (shortcut).

```

Step 4. Perform the balanced binary tree prefix-sum computation described in the previous section with respect to the vector $RULING$. As a result:

(1) $m := \sum_j RULING(j)$, and

(2) each element j with $RULING(j) = 1$ gets its entry number in a (contracted) array of length m containing the output linked list.

(This array is the input for the next iteration (if any) of the whileloop.)

od

Let T_1 (resp. T_2) be the first (resp. last) time unit for which an assignment into $OP(,)$ was performed.

Step 5. Apply a simulation of the standard deterministic algorithm by p processors to the current array.

Step 6.

```

for Processor  $i$ ,  $1 \leq i \leq p$ , pardo
  for  $t := T_2$  downto  $T_1$  do
     $R(OP(i,t).1) := R(OP(i,t).2) - OP(i,t).3$  .
    (Comment.  $OP(i,t).k$ ,  $k=1,2,3$ , represent the fields of  $OP(i,t)$ . Also, recall Comment (c) in the verbal description of Step 3.)

```

Implementation remark. Each time m gets a new value, broadcast it to all processors as in Implementation Remark 1 of the previous section.

Complexity

Initialization requires $O(n)$ operations and $O(1)$ time. Let us focus on one iteration of the while loop.

Step 1 takes $O(m)$ operations and $O(1)$ time.

Step 2 takes $O(m \log^* m)$ operations and $O(\log^* m)$ time.

Step 3 takes $O(m)$ operations and $O(1)$ time.

Step 4 takes $O(m)$ operations and $O(\log m)$ time.

So each iteration of the while loop takes $O(m \log^* m)$ operations and $O(\log m)$ time. Each such iteration results in a linked list whose length is $\leq 1/2$ the

length of the list when the iteration started. Therefore, after $O(\log \log n)$ iterations we get a list whose length is $\leq n/\log n$. Summing up the operation and time complexity of the **while** loop gives $O(n \log^* n)$ operations and $O(\log n \log \log n)$ time.

Step 5 takes $O(n)$ operations and $O(\log n)$ time.

Step 6 requires the same number of operations and time as all the iterations of Step 3, since it follows its "footsteps".

So we got a total of $O(n \log^* n)$ operations and $O(\log n \log \log n)$ time. Applying Brent's theorem we get $O((n \log^* n)/p)$ time using any number $p \leq (n \log^* n)/(\log n \log \log n)$ of processors. We know that any such result can be alternatively stated as $O(\log n \log \log n)$ time using $(n \log^* n)/(\log n \log \log n)$ processors. We leave the reader to verify that the implementation problems as per the remark following Brent's theorem can be readily overcome.

5. The Optimal Algorithm

We describe an algorithm that runs in time $O(nk/p + \log^{(k)} n \log n + k \log n)$ using any number p of processors, for $k \leq \log^* n$ (henceforth, we assume $k \leq \log^* n$). We deduce the following results.

- (1) For fixed k , with $p = n/(\log n \log^{(k)} n)$ we achieve a running time of $O(k \log n \log^{(k)} n) = O(\log n \log^{(k)} n)$; this is optimal speed-up.
- (2) For $k = \log^* n$, with $p = n/\log n$, we achieve a running time of $O(\log n \log^* n)$, since for $k = \log^* n$, $\log^{(k)} n \leq 2$.

A variant of the algorithm will yield our third result.

The basic algorithm (of the previous section) had two stages. In the first stage (the **while** loop) we employed an almost optimal algorithm (given a list of length m it performed $O(m \log^* m)$ operations). In the second stage (step 5) we used an algorithm that performed relatively more operations (for a list of length m , $O(m \log m)$ operations), but it had the advantage of being faster. To profit from this we needed to ensure that the numbers of operations performed by the two stages were roughly the same. And, in fact, this was the case, because the list processed in the second stage was sufficiently shorter. Our present algorithm pushes this methodology further. The algorithm has three main stages, each one processing a relatively shorter list. Stage 1 uses an optimal algorithm that is relatively slow; its effect is to slightly reduce the length of the input list. Stage 2 uses an almost optimal algorithm; it is faster. Its effect is to further reduce the length of the list. Stage 3 uses the standard deterministic algorithm. It misses optimality by a logarithmic factor, but it is the fastest of the three algorithms. The overall result is a fast optimal algorithm. This methodology was also used in [Vi-83b].

The input for Stage 1 is the input linked list of length n . The output of stage 1 (and input for Stage 2) is a linked list of length $\leq n/(\log^{(k+1)}n)^3$. The output of Stage 2 (input for Stage 3) is a linked list of length $\leq n/(\log n)^3$. Each of the linked lists mentioned above is given in an array whose size is the same as the length of the list. Stage 3 simply consists of applying the standard deterministic parallel algorithm.

Remarks. The algorithm will be described in less detail than the preceding algorithms. In particular:

1. At each timestep of stages 1 and 2 we have a linked list that was obtained from the input list by propagating pointers over vertices that were omitted (as in the previous section). In particular, every edge, in any of the linked lists that are obtained throughout these stages, corresponds to a directed path in the original input list. We must maintain a vector (like R in the previous section) that holds, for each such edge, the length of its original path. However, in this presentation we focus only on the transitions from a given linked list to a shorter one and avoid mentioning updates of this vector.
2. Note that in (stages 1 and 2) we only mentioned contractions of a linked list into a shorter one (the up-sweep part using the term of Section 3). We will systematically omit the corresponding down-sweep part throughout this section. No new ideas (beyond Section 4) are required in order to fill in this part.

Stage 1. This stage employs Procedure 1 repeatedly.

Procedure 1.

Input: A linked list of length m given in an array of length m .

Output: A linked list of length $\leq m/2$ given in an array of the same length as the list.

Procedure 1 proceeds as follows.

1. Apply the basic step (of Section 2) $k+1$ times to obtain a $\log^{(k+1)}m$ -ruling set. (Denote the cardinality of this ruling set by m_1).

Explanation. The output list of Procedure 1 will consist of the vertices of the ruling set. So for each vertex v in the ruling set the remaining job for Procedure 1 is to traverse the input list up to the first successor that is also in the ruling set (to be called the *sublist* of v). (Recall that the edge length of each such sublist is between 2 and $\log^{(k+1)}n + 1$.) This remaining job might cause difficulties if we used a naive assignment of processors to their jobs as per the remark following Brent's theorem (particularly if optimal speed-up is desired). Below, we show how to overcome these difficulties.

2. Using the prefix sum algorithm assign numbers from 1 to m_1 to the vertices of the ruling set. (Each vertex represents its sublist, which is thereby implicitly numbered.)

Conceptually, in stages 3 and 4 below, we partition the work of traversing the m_1 sublists among $m/(\log m \log^{(k+1)}m)$ processors. We have two phases.

3. Phase 1 consists of time pulses. At each time pulse each processor is "in charge" of $\log m$ sublists. For each of its sublists the processor advances down one edge; if an element of the ruling set is not encountered (meaning that the traversal of the sublist is not yet completed) then the processor remains in charge of the sublist. At the end of each pulse each processor needs to acquire a few new sublists in order to restore its number of sublists to $\log m$. The situation is as follows:

- (1) Each sublist, up to sublist q , for some $q < m_1$, has been previously assigned to a processor.
- (2) Processor i , $0 \leq i < m/(\log m \log^{(k+1)} m)$, needs a_i additional sublists to restore its number of sublists to $\log m$.

In parallel, the processors perform a prefix sum computation on a_i (in time $O(\log m)$). Let $b_i = \sum_{j=0}^{i-1} a_j$, for $0 \leq i \leq m/(\log m \log^{(k+1)} m)$. Processor i acquires a_i new sublists by taking the sublists $q + b_{i-1} + 1$ through $q + b_i$. (This makes sense only as long as we refer to indices $\leq m_1$.) If $\sum_{j=0}^{m/(\log m \log^{(k+1)} m)} a_j \geq m_1$ then all the sublists have been assigned to processors

and we proceed to phase 2. Otherwise, another pulse of Phase 1 is performed. (This application of a prefix sum computation is very similar to the (known) use of the primitive Fetch-and-Add by the NYU-Ultracomputer for the parallel implementation of a queue (see [GLR-83]).)

4. Phase 2. The situation is that each of the $m/(\log m \log^{(k+1)} m)$ processors is in charge of at most $\log m$ sublists, where the length of each sublist is $\leq \log^{(k+1)} m$. Each processor simply completes the traversal for all its sublists.

5. A prefix sum computation is applied in order to contract the input array into an array of size m_1 containing only the vertices of the output linked list.

Time complexity of Stage 1.

Complexity of an iteration of Procedure 1.

Step 1: $O(k)$ time, $O(mk)$ operations.

Each of steps 2 and 5: $O(\log m)$ time, $O(m)$ operations.

Step 3: Recall that we use $m/(\log m \log^{(k+1)} m)$ processors. In each pulse each processor needs $O(\log m)$ time to traverse one edge in each of its sublists. An additional $O(\log m)$ time is needed for the prefix sum computation. Since each pulse provides for traversals of $m/\log^{(k+1)} m$ edges there will be at most $\log^{(k+1)} m$ pulses before the queue is empty and we proceed to Step 4. So Step 3 takes $O(\log m \log^{(k+1)} m)$ time and $O(m)$ operations.

Step 4: $O(\log m \log^{(k+1)} m)$ time using $O(m)$ operations.

So an iteration of Procedure 1 takes $O(\log m \log^{(k+1)} m)$ time and $O(mk)$ operations.

Complexity of Stage 1. The output of Stage 1 is a linked list whose length is $\leq n/(\log^{(k+1)}n)^3$. Since at each invocation of Procedure 1, $m_1 \leq m/2$, we need to use at most $3\log^{(k+2)}n$ iterations of Procedure 1. So the total number of operations in Stage 1 is $O(nk + nk/2 + nk/4 + \dots) = O(nk)$ and the total time is $O(\log n \log^{(k+1)}n \log^{(k+2)}n) \leq O(\log n \log^{(k)}n)$.

Stage 2. Stage 2 consists of k iterations of Procedure 2.

Iteration i of Procedure 2, ($1 \leq i \leq k$).

Let $j = k + 1 - i$.

Input. A linked list of length at most $n/(\log^{(j+1)}n)^3$, given in an array having the same length as the list.

Output. A linked list of length at most $n/(\log^{(j)}n)^3$, given in an array having the same length as the list.

1. Apply $3\log^{(j+1)}n - 3\log^{(j+2)}n$ iterations of Routine 1.

Iteration g of Routine 1, $0 \leq g < 3\log^{(j+1)}n - 3\log^{(j+2)}n$.

Input. A linked list of length $m \leq 2^{-g}n/(\log^{(j+1)}n)^3$, given in an array of length $\leq n/(\log^{(j+1)}n)^3$. (The vertices of the linked list are "spread over" the array which may have more entries than the length of the list. Redundant entries of the array (i.e., entries that represent vertices which are not in the input list for iteration g) are marked as such. The reason for this "wasteful" representation of the input is that iterations of Routine 1 "save time" by not contracting their input array to include only their output list. Only the end of Procedure 2 contracts the linked list at hand).

Output. A linked list of length $m_1 \leq m/2$, given in an array of length $\leq n/(\log^{(j+1)}n)^3$.

(a) Apply the basic step (of Section 2) $j+1$ times to obtain a $\log^{(j+1)}n$ -ruling set. (Denote the cardinality of this ruling set by m_1).

Explanation. The output list of the present iteration of Routine 1 will consist of the vertices of the ruling set. So for each vertex v in the ruling set the remaining job is to traverse the sublist of v . (Recall that the edge length of each such sublist is between 2 and $\log^{(j+1)}n + 1$.)

(b) **for** $1 \leq v \leq n/(\log^{(j+1)}n)^3$ **pardo**

if $v \in$ the ruling set

then traverse the sublist of v .

This completes iteration g of Routine 1.

Step 2 below concludes the present iteration of Procedure 2.

2. A prefix sum computation is applied in order to contract the input array into an array containing only the vertices of the linked list at hand.

Time complexity of Stage 2.

Complexity of iteration g of Routine 1: Using $n/(\log^{(j+1)}n)^3$ processors, Step (a) takes $O(j)$ time and Step (b) takes $O(\log^{(j+1)}n)$ time. This yields a bound of $O(nj/(\log^{(j+1)}n)^2)$ operations taking $O(\log^{(j+1)}n + j)$ time.

Complexity of iteration i of Procedure 2: Step 1 consists of $O(\log^{(j+1)}n)$

invocations of Routine 1. Step 2 needs $O(n/(\log^{(j+1)}n)^3)$ operations and $O(\log n)$ time. Thus the i -th iteration of Procedure 2 performs $O(jn/\log^{(j+1)}n)$ operations in time $O((\log^{(j+1)}n + j) \cdot \log^{(j+1)}n + \log n) = O(\log n)$.

So, overall, Stage 2 performs $O(\sum_{j=1}^k \frac{jn}{\log^{(j+1)}n}) \leq O(kn)$ operations in time $O(k \log n)$.

Stage 3 requires $O(\log n)$ time and $O(n/\log^2 n)$ operations. It is also easy to bound the time and number of operations required by the down-sweep part (which is missing in the above description) by the same time and number of operations as for stages 1 and 2.

Putting everything together, and applying Brent's theorem, we deduce the algorithm runs in time $O(nk/p + \log^{(k)}n \log n + k \log n)$, using any number p of processors, where $k \leq \log^* n$. The implementation problems as per the remark following Brent's theorem can be readily overcome.

Remark: We can also obtain a class of algorithms taking time $O(\log n)$ on $n \log^{(k)}n / \log n$ processors, for any fixed positive integer k , as follows. By way of motivation, we observe that, in the algorithm just described, stage 2 is faster than stage 1 (on equal length inputs), but requires more operations. Therefore, by substituting stage 2 for stage 1, we might expect to reduce the running time and increase the total number of operations. So, in the above algorithm, we replace Stage 1 with Routine 1 applied $3 \log^{(k+2)}n$ times, where the input for the g -th iteration is a linked list of length $\leq 2^{-g}n$, stored in an array of length n ; also, in part (a) of the routine, we seek a $\log^{(k+1)}n$ -ruling set. Then we perform the rest of the above algorithm with no change. We achieve a running time of $O(k \log n)$ taking $O(n \log^{(k+1)}n \log^{(k+2)}n + kn) \leq O(n \log^{(k)}n)$ operations. Our result follows by Brent's theorem. This shows that Wyllie's conjecture which was mentioned in the introduction is not correct.

6. Open problems

- (1) Is there an optimal speed-up algorithm for the list ranking problem using $n/\log n$ processors and running in time $O(\log n)$?
- (2) We recall that the new coin tossing technique distinguishes the PRAM model from the more abstract PRAM-INFINITY model. We are not aware of any other technique having this property. Are there others? In addition, this remark calls for a "metatheoretical" discussion of the applicability of PRAM-INFINITY lower bounds to PRAMs. We note that a lower bound in the PRAM-INFINITY model is stronger than the same lower bound in the decision tree model, a model that is often used when proving lower bounds. Also, non-trivial lower bounds have been proved for the PRAM-INFINITY model. Thus it seems useful to ascertain the applicability and limitations of such lower bounds.

Acknowledgement Much gratitude to Baruch Schieber for helpful discussions and comments and to Dennis Shasha for useful comments on an earlier draft.

7. References

- [AHU-74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [CLC-81] F.Y. Chin, J. Lam and I. Chen, "Optimal parallel algorithms for the connected component problems," *Proc. 1981 International Conf. on Parallel Processing* (1981), 170-175.
- [FMRW-85] F.E. Fich, F. Meyer auf der Heide, P. Ragde and A. Wigderson, "One ,two, three...infinity: lower bound for parallel computation", *Proc. 17th Annual ACM Symp. on Theory of Computing* (1985), 48-58.
- [GLR-83] A. Gottlieb, B.D. Lubachevsky and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors", *ACM TOPLAS*, 1983, 164-189.
- [IM-85] A. Israeli and S. Moran, private communication.
- [KRS-85] C.P. Kruskal, L. Rudolph and M. Snir, "The power of parallel prefix", *Proc. 1985 International Conf. on Parallel Processing*, 180-185.
- [MW-85] F. Meyer auf der Heide and A. Wigderson, "The complexity of parallel sorting", *Proc. 26th IEEE Annual Conf. on Foundations of Computer Science* (1985), to appear.
- [TV-83] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm", TR 69, Dept. of Computer Science, Courant Institute, NYU, 1983. To appear in *SIAM J. Comput.*
- [Vi-83a] U. Vishkin, "Synchronous parallel computation - a survey", TR 71, Dept. of Computer science, Courant Institute, NYU, 1983.
- [Vi-83b] U. Vishkin, "An optimal parallel algorithm for selection", manuscript, 1983.
- [Vi-84a] U. Vishkin, "An optimal parallel connectivity algorithm", *Discrete Applied Math.* 9 (1984), 197-207.
- [Vi-84b] U. Vishkin, "Randomized speed-ups in parallel computation", *Proc. 16th Annual ACM Symp. on Theory of Computing* (1984), 230-239.
- [Vi-85] U. Vishkin, "On efficient parallel strong orientation", 1984, *Information Processing Letters* 20 (1985), 235-240.

- [W-79] J.C. Wyllie, "The complexity of parallel computation," TR 79-387, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

This book may be kept

FOURTEEN DAYS

A fine will be charged for each day the book is kept overtime.

GAYLORD 142			PRINTED IN U.S.A.

NYU COMPSCI TR-175 C1
Vishkin, Uzi
Deterministic coin tossing
with applications to
optimal parallel list

NYU COMPSCI TR-175 C1
Vishkin, Uzi
Deterministic coin tossing
with applications to
optimal parallel list

LIBRARY
N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012

